A REAL-TIME DATABASE SYSTEM FOR MANAGING AQUARIUM DATA

A Thesis
by
DEVIN SINK

Submitted to the Graduate School
Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

August 2017
Department of Computer Science

A REAL-TIME DATABASE SYSTEM FOR MANAGING AQUARIUM DATA

A Thesis
by
DEVIN SINK
August 2017

APPROVED BY:

_____
Rahman Tashakkori, Ph.D.
Chairperson, Thesis Committee

_____
James B. Fenwick Jr., Ph.D.
Member, Thesis Committee

_____
Raghuveer Mohan, Ph.D.
Member, Thesis Committee

_____
Rahman Tashakkori, Ph.D.
Chairperson, Department of Computer Science

_____
Max C. Poole, Ph.D.
Dean, Cratis D. Williams School of Graduate Studies

Abstract

A REAL-TIME DATABASE SYSTEM FOR MANAGING AQUARIUM DATA

Devin Sink
B.S., Appalachian State University
M.S., Appalachian State University

Chairperson: Rahman Tashakkori, Ph.D.

Relational databases are not ideal for monitoring systems that are continuously changing because of their slow response times. A real-time NoSQL database allows for faster response times due to receiving data in real-time rather than having to poll the database for changes. This thesis creates a real-time NoSQL database for the purpose of monitoring an aquarium and conducts analysis on the advantages and disadvantages over a relational database.

Acknowledgments

First I would like to thank my advisor, Dr. Rahman Tashakkori, for being my mentor throughout my time here at Appalachian State University. Without his encouragements I wouldn't have thought graduate school was possible.

Next I would like to thank my committee members for taking the time to read the thesis and provide valuable feedback.

Finally I would like to thank my parents, for always pushing me to excel in anything that I do.

Contents

List of Figures

Chapter 1 - Introduction

1.1 Statement of the Problem

Relational databases are not suitable for the needs of a system in which real-time data and observations are important and timely decisions are critical. For any monitoring system, real-time data collection means improving the success rate of catching and correcting problems. For example, consider a system tasked with monitoring the pressure within industrial equipment. If the pressure goes too high, catching and correcting it early could be the difference between a delay in production and total system failure. A real-time database allows users to receive notifications about their systems more reliably than polling a relational database. Polling limits a user to receive new data based on how often they are polling the database. This is not the case with a real-time database, as the user would receive notifications as they occur, eliminating the need to constantly query the database.

1.2 Goals

The overall goal of this thesis is to build upon an existing relational database to achieve a real-time feature. The current system provides data by constantly querying the database. This provides data that is not real-time. A real-time database would allow for changes to be pushed out to the website of the system as they are entered into the database. The

database holds temperature, pH, and ambient humidity readings, as well as outlet states, recorded from probes that are fixed to an aquarium. The introduction of a real-time component would improve how user notifications, graphs, and data panels work within the website. For example, a user viewing the home page for his/her aquarium can see the displayed notifications associated with that aquarium. Should there be an alert sent to tell the user that the temperature is too high, they will not actually see that notification until they refresh the page. In such a case, the controller in place will handle the problem, but the user will be unaware if he/she has not checked the notifications yet. With the live updating nature of a real time database, the notification would be pushed out as it is entered into a notification table, causing the notification panel to update.

This thesis will compare the performance of basic database operations by building two NoSQL databases to evaluate alongside the existing relational database. The two NoSQL databases used will be RethinkDB and MongoDB. Because RethinkDB is so new, there is very little or no evaluation data available on its performance. MongoDB is added during the testing as a benchmark for NoSQL performance since numerous performance evaluations have been recorded for it. The testing is done to see if RethinkDB has the potential to be the sole database supporting a monitoring system, or whether it should be a hybrid of RethinkDB and MySQL.

Chapter 2 - Literature Review

2.1 Relational Databases

A relational database is a collection of related tables, each with a rigid structure, related to one another through the use of unique keys or identifiers. An example of a database would be a simple user table that contains a unique id, username, email address and password. These defining characteristics are referred to as attributes. Alongside this table could be a billing information table, that contains credit card numbers, expiration dates, etc. To relate these two tables, the unique user id would be placed in the record with the corresponding card in order to associate it to its owner. The rigid structure means that the table has to be fully created and all attributes have to be defined before adding any data.

The relational data model has been the standard for a majority of database systems for many years. A relation in this case is a table, and a relational database is a collection of these tables that are efficiently connected using the unique key of each respective table. The rows of each table represent individual entries, or records. The columns of each table are called attributes, as they describe the records. For example, a sample customer table may have columns/attributes representing a unique customer id, name, and email. Each record represents a unique customer [1].

MySQL is a popular relational database. Williams et al. [2] go into great detail in their book explaining how to use MySQL coupled with PHP [3] to create a database driven website. They show how databases are used on the web by explaining the three-tier architecture model. The three tiers are a client, a web server, and a database. A client makes a request to a web server, a response is generated from a database, then that data is served back to the client. Data is accessed from the database by using SQL (Structured Query Language).

Atzeni et al. [4] argue that relational databases are past their prime, and unsuitable for today's growing data management needs. Relational databases were designed for data management that was geared towards administrative applications. The purpose of relational database systems was to facilitate well structured data like accounts, loans, and various other facets of banking. But today's data management applications need to handle unstructured or complexly structured data, including the capability to store streaming data, documents, and graphs; mainly things that do not fit very well into a rigid relational database. In addition, some data and their management may come from various sources demanding real-time interactions and updates.

2.2 NoSQL Databases

NoSQL databases are collections of related tables, but they do not have to have a defined structure like a relational database does. It is possible to edit records and append at-

tributes without compromising the rest of the data in the table. In the previous example with the user table, deciding to add a flag to a user to identify them as an admin is possible without having to add an admin column for each individual user. For example, in a traditional relational database, there would have to be an admin column with a 1 to signify this user was an admin, and all other users would have a 0 attached to them. With NoSQL, this is not necessary. One can add attributes per user without taking up unnecessary space in the table.

Bigtable is a scalable NoSQL database from Google that serves as the foundation for multiple other NoSQL platforms, particularly HBase and Cassandra [5]. Bigtable utilizes column families, which are column keys that have been grouped into sets. All the columns in a column family are related, meaning that a column family is analogous to a table within a relational database. Bigtable is used in a number of Google applications, such as Google Maps and Google Earth. Within these applications, the system uses one table to preprocess data and store raw images, and a second set of tables to deliver data to the client.

HBase is an open source version of Bigtable that is written in Java. Carstoiu et al. [6] evaluate the reading and writing performance of HBase by recording the number of operations per second using various row counts. Their methodology is as follows: First, a table named "test" is created with a single column, then rows containing randomly generated values are inserted with random row ids. Then this table is deleted, and another

single column table named "test" is created, this time inserting randomly generated values with sequential row ids. Using this table, an amount of reads equal to the amount of records are performed first sequentially then randomly. Finally, a series of updates are performed first sequentially then randomly. They perform these steps with 6 different row counts: 10,000, 100,000, 300,000, 500,000, 700,000, and 1,000,000. Carstoiu et al. were able to conclude the following: sequential inserts were marginally faster than random inserts, writing new data was at its fastest at the 500,000 row mark, updating rows (both sequentially and randomly) was faster than inserting rows on counts of up to 300,000, and reading rows (both sequentially and randomly) was at its fastest with lower row counts, with speed decreasing as row count increased.

Prasad et al. [7] briefly covered a number of NoSQL databases by first outlining their common characteristics, then comparing their performance based on four factors: replication, sharding, consistency, and partial fault tolerance. Replication in this context means automatically copying data to another machine, which aids in maintaining availability through outages, failures, etc. Sharding is horizontal scaling that spreads data across multiple servers, allowing operations on larger sets of data without affecting response times. Consistency refers to the inclusion of concurrency control or transaction management to guarantee correctness. Fault tolerance refers to built-in mechanisms to deal with any failures. The database platforms covered are SimpleDB, Dynamo, Cassandra, Riak, HBase, Redis, CouchDB, MongoDB, and Bigtable. Their results conclude

that all examined platforms supported replication and sharding in some fashion. Four of the platforms, SimpleDB, Cassandra, Riak, and CouchDB, do not have any methods of enforcing consistency. Two of the platforms, SimpleDB, and Dynamo, do not have any failure handling.

Srivastava et al. [8] studied a few of these platforms, particularly MongoDB and Cassandra, a little more closely, pointing out individual strengths and weaknesses between them. MongoDB is a document store database that stores semi-structured JSON encoded data. The authors report that MongoDB is best suited for content management of semi-structured data and a replacement for relational databases when building web apps. It is not suited for applications needing multiple join operations or foreign key references. Cassandra is an open-source column oriented database, similar to Bigtable. Cassandra is recommended for applications that execute more reads than writes, Twitter, for example. It is also recommended for applications that provide dynamic content to users, like the backend of Netflix. Cases where Cassandra is not recommended are applications where high consistency is required because Cassandra sacrifices some data consistency in favor of high availability.

One of the common characteristics mentioned above is lack of required schema for a NoSQL database. Unlike traditional databases, data can be freely inserted without defining a rigid schema. As a result, the data model of a NoSQL database can differ greatly from a relational database. Hecht et al. [9] break this down into four distinct

groups: key value stores, document stores, column family stores, and graph databases. Key value stores are similar to mapped values, where each entry is addressed by a unique key. Document stores are key value pairs in JSON. Unlike key value stores, where the value is only accessible by key, values within document stores can be queried as well. Column family stores are also known as column oriented stores, in which data is stored by column rather than by row. Graph databases represent heavily linked data. One example given for a graph database is FlockDB, which manages relationships between Twitter users to facilitate following another user. Querying this data also differs greatly from traditional SQL. Kaur and Rani outline a few different querying formats [10]. Since it is so new, there is no standard query language for use with non-relational databases. Querying tends to be data-model specific, or vary by platform. Most of the NoSQL databases allow for RESTful interfaces to interact with data, while some offer their own query API. For example, RethinkDB has ReQL, Cassandra has CQL, etc.

2.3 Real-time Databases

For the purposes of this thesis, a real-time database adds an additional functionality on top of a NoSQL database. When the database changes state, the changes are recorded and pushed out to anything monitoring the database. This is significant from a users perspective because they receive any changes in real-time as they occur.

RethinkDB is an open-source document-store database that utilizes what are called *changefeed*s to continuously push database changes in real time. This eliminates the need to have users continuously polling data from the database. RethinkDB has its own query language, called ReQL, which constructs queries in a similar manner to stringing together function calls. Instead of utilizing the locking method of concurrency control, RethinkDB implements multi-version concurrency control. This means that each time data is set to be updated, it is not overwritten; instead, a new version is created and inserted. This ensures that attempting to write data will not interrupt any ongoing reads [11].

In addition to the requirements of a conventional database, real-time databases can have the additional responsibility of completing transactions before system defined deadlines. Scheduling these transactions is a way to ensure a system's timing constraints are met. Kao et al. [12] outline a few aspects of transactions in addition to deadlines that are taken into consideration when scheduling. Most notably, the criticalness and the value function of a transaction. Criticalness is simply a measurement of how critical it is that a transaction meets its deadline. Criticalness and deadlines are conceptually different; it is possible that a transaction can have a short deadline, but missing that deadline has a low impact on the system. This is further explained when examining the value function of a transaction. The value function shows how valuable to the system it is to complete a transaction after the deadline has been passed. The authors describe four

example value functions. The first case is diminishing positive value, where completing a transaction is beneficial even when late. However, the net positive impact decreases as more time passes. The next case is zero value, where once the deadline has been passed, the transaction no longer has value to the system. The third case shown is negative value, where missing the deadline will have a negative impact on the system, but that negative impact remains constant as time passes. The last case shown is increasingly negative value, where missing the deadline of a transaction will have a negative impact, and that negative impact grows as more time passes. Using the deadline, criticalness, or both, a priority is assigned to transaction.

A scheduler's goal is to execute transactions concurrently, while having the end result be identical to executing those same transactions sequentially. Ensuring correct results at the end of these transaction blocks is referred to as concurrency control. There are two basic views to concurrency control: pessimistic, where one assumes many transactions will conflict with each other, and optimistic, where one assumes there will be few conflicting transactions. Most well known pessimistic concurrency control methods are based on two phase locking (2PL). The two phases being the growing phase, where locks are acquired but not released, and the shrinking phase, where locks are released but not acquired. The 2PL approach guarantees serializability by blocking multiple transactions from accessing the same data. Lindstrom [13] presents a variation of 2PL titled 2PL High Priority. In this variation, high priority transactions can bypass locks applied by

lower priority transactions. For example, if a mid priority level transaction has placed a lock on specific data, but a high priority level transaction then requests the same data, the mid-level transaction will be aborted and rolled back, granting the lock to the higher priority transaction. Inversely, if a low priority level transaction requests the data in the scenario above, the low level transaction is blocked to wait for the lock holder to release.

Barbosa [14] outlines two more scheduling policies known as rate monotonic and earliest deadline first. With rate monotonic scheduling, each transaction has a fixed priority level that is inversely proportional to its period. So transactions with the shortest periods have the highest priority. Rate monotonic is an optimal fixed priority scheduling algorithm, meaning any given set of transactions that can be scheduled with fixed priority, can be successfully scheduled using the rate monotonic algorithm. Barbosa's second scheduling policy is earliest deadline first, meaning the transaction with the closest deadline will have the highest priority. The earliest deadline first algorithm assigns priority dynamically, meaning priorities are calculated and adjusted as needed during execution. While rate monotonic scheduling is recommended for systems with periodic transactions, earliest deadline first can handle both periodic and non-periodic transactions.

Systems with timing constraints are the main application for real time databases. Mall [15] outlines the use of real time databases in spacecraft control systems. Onboard control systems such as these are responsible for monitoring the status of the spacecraft. While the volume of this status information (temperature, acceleration, etc.) is relatively

small, the timely delivery of it is imperative. Lam et al. [16] echo this, expanding into general avionics, as well as air traffic control. One critical part of air traffic control is consistently and accurately maintaining weather data. In the event of dangerous weather, the controller needs to be made aware of it immediately, so traffic can be routed accordingly.

2.4 Relational vs NoSQL

Mohamed et al. [17] analyze the differences between relational and NoSQL databases on a conceptual level. A few of the areas they touch on are scalability, handling big data, and complexity. Relational databases are vertically scalable, meaning they can only be improved by adding more hardware resources to a single machine. This restriction in scalability is what holds the relational databases back from performing well on cloud platforms and handling big data. NoSQL databases however, are horizontally scalable, meaning more machines can be added, reducing the overall workload on a single machine. Splitting up tasks across multiple machines allows NoSQL databases to perform transactions on incredibly large data sets. The authors also note a potential for higher complexity when working with relational databases because data must first be converted into tables that fit a particular structure, whereas NoSQL databases have the capability to store data regardless of structure. Priyanka [18] reiterates these same differences and

adds that NoSQL databases support partition tolerance, while relational ones do not. This means NoSQL databases will continue to function even if one of the nodes fails.

Bhogal et al. [19] expand on using both relational databases and NoSQL databases to handle big data by building a sample database of each. The database template for both is a library inventory system. For the relational database, the authors selected Oracle Application Express (APEX). APEX is a development environment running on top of an Oracle database. For the NoSQL database, the authors selected MongoDB. After building both databases and inserting various products, they were able to confirm two of the previous criticisms. Since attributes of products can be listed across various tables, complex joins were required in the relational database to display all attributes of a product. This was not the case in NoSQL databases, as all attributes were obtainable with a single query. The limitations of scalability were also noted, stating that most new data is unstructured, making it impossible to incorporate all data types. Again, this is something easily handled by NoSQL databases.

Parker et al. [20] performed a similar test, analyzing the querying speed of inserts, updates, selects, and aggregate functions between MongoDB and a database built with SQL. The sample database in this scenario consisted of three tables: user, department, and project. Testing the insert speed of either database platform was inconclusive. Neither platform was consistently faster than the other. When updating, MongoDB was faster in all cases that updated using the primary key. However, it was consistently

slower in updating a table using a non-indexed column. For select statements, MongoDB outperformed SQL in nearly all cases. The authors make note that this is likely due to MongoDB storing things in memory rather than on disk. Queries making use of aggregate functions were the only category in which SQL was strictly better than MongoDB.

MonBench is a benchmarking tool used to generate workloads to simulate heavy conditions. Zhao et al. [21] ran this tool on three separate databases as a way to test query times.The three databases used were a traditional relational MySQL database, a non-relational distributed database, HBase, and OpenTSDB, a time series database using a cache mechanism. Using MonBench, they were able to determine that MySQL has the slowest overall query time. In common cases without using the cache, HBase was marginally faster than OpenTSDB. However, OpenTSDB's query response time was dramatically reduced when the data was available in the cache.

Chapter 3 - Methodology

The development of this thesis has three phases. The first phase includes a relational database built with MySQL, and then its equivalent database for RethinkDB. The second phase is a series of web components used to monitor the data from each database. These components are a data panel, a notification table, and a data graph. These components will be outlined in section 3.2. The third phase consists of recreating a previously mentioned database performance test. The cited test was performed on HBase alone. This thesis recreates it to analyze MySQL, MongoDB, and RethinkDB. Further analysis was performed using a monitoring tool named Datadog [22] to see the impact of these actions on the hardware running these tests.

3.1 Databases

3.1.1 Relational Aquarium Database

The MySQL aquarium database, as shown in figure 3.1, consists of seven tables:

- Alerts- This table serves as a legend for defining what a specific alert id means.
- Alert Log- This table is a collection of alert ids, Arduino ids, and a timestamp indicating what happened, what Arduino recorded it, and at what time.

- Aquarium- This table contains a unique id for each aquarium, as well as listing what user owns it, what Arduino is assigned to it, and a user defined aquarium name and type.

- Arduino- This table contains a unique id for each Arduino, a name for the Arduino, maximum and minimum boundaries for recorded parameters, and on/off times for each outlet it can control.

- Data- This table is where the Arduino writes the data it records. The parameters being water temperature, pH, ambient temperature, ambient humidity, four outlet states, and a timestamp.

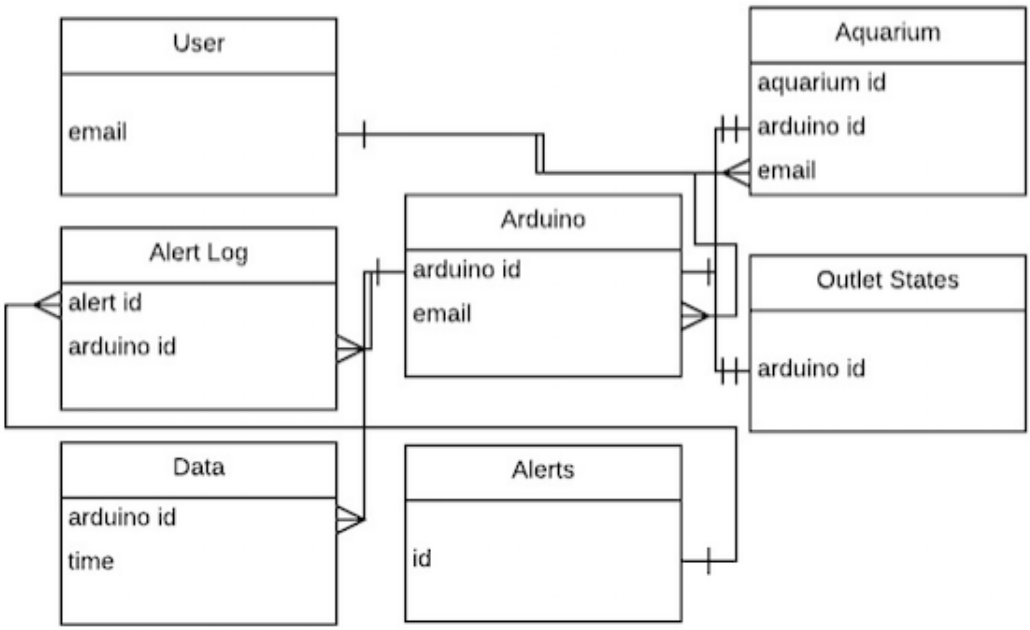


Figure 3.1: Aquarium Database Schema

- Outlet States- This table shows the current state of four outlets (either on or off).
- User- This table is a collection of all the users on the website. For each user, an email, password, and username are recorded.

A website was made in order to easily access the data within the database. The website that houses these PHP components also allows for user registration, adding/editing of user aquariums, and adding/editing of user Arduinos. When adding an aquarium, the user enters a name and type (i.e. salt or fresh water) of their choosing. The user also has the option to link an Arduino [23] that isn't already associated with an aquarium. An Arduino is a microcontroller with probes in the aquarium for the purpose of recording data. This association implies that this specific aquarium is being monitored by a specific Arduino. When adding an Arduino, the user enters a name for the Arduino, and can optionally set bounds(max and min) for their temperature and pH. The user can also name up to four outlets the system controls, and provide on and off times for each to set a schedule. For example, the user can name outlet 1 "Lights" and schedule them to turn on at 8AM then turn off at 4PM. Additionally, the user can choose to link an Arduino to a currently unmonitored aquarium. Figure 3.2 shows an example of adding a new aquarium with no unused Arduinos available.

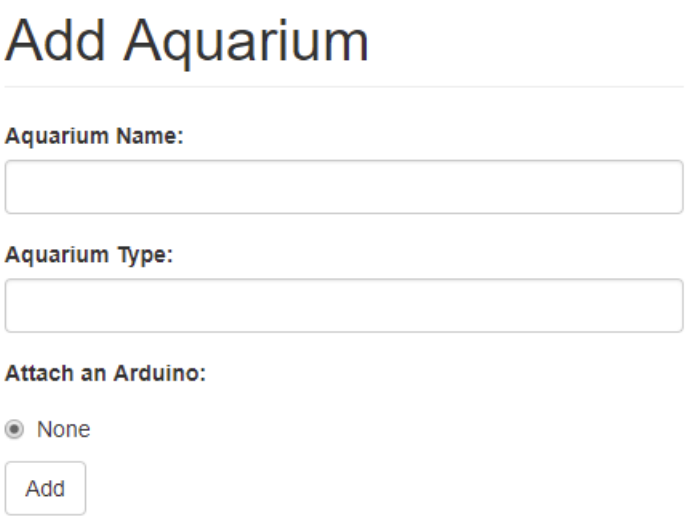


Figure 3.2: Add Aquarium Form

3.1.2 Real-Time Time Aquarium Database

The RethinkDB aquarium database only has two tables. The goal of this thesis is to improve the performance of delivering data back to the user. The rest of the user management (adding/editing aquariums and Arduinos, etc.) is left for MySQL to handle. Figure 3.3 shows an example of how one row of data is represented in the RethinkDB real-time database.

- Alerts- This table serves as a combination of the MySQL Alerts and Alert Log table. Recording a relevant alert message, and at what time the alert occurred.

- Data- This table is similar to the MySQL Data table, recording water temperature, ambient temperature, ambient humidity, pH, and time.

```
{
    "ambient humidity": "29.7999992371" ,
    "ambient temperature": "68.0" ,
    "arduino id": "4" ,
    "id": "0005ae16-f0a3-4348-abd1-a308c4ba538c" ,
    "ph": "0" ,
    "temperature": "77.7866" ,
    "time": 1489455843

}
```


Figure 3.3: View of how data is stored in RethinkDB

3.2 Web Components

3.2.1 Data Panel

The first web component is a data panel. The data panel serves as a log of entries showing past readings of a users aquarium. One entry shows the time it was recorded, the temperature of the water in the aquarium, as well as the temperature and humidity of the ambient air around the aquarium. The first version of this panel, based on the relational database, was static and did not change after the page had loaded. The new version updates automatically and inserts the new entry into the list. This is achieved by monitoring the *changefeed* of the database in RethinkDB. Whenever a new entry is recorded, changes to the table are sent out to subscribers to the *changefeed*.

The first version of the panel is written in PHP. When the page loads, it queries the MySQL database to grab the most recent entries by using the *mysql_query* func-

tion. Within the HTML producing the panel, there is a PHP block that prints out each record wrapped in table tags to format it. The second version of the panel is written in JavaScript [24]. JavaScript is one of the languages listed as officially supported by RethinkDB, so it was chosen to simplify the creation process. Like the PHP version, the most recent entries are fetched from the database and added to an array when the page first loads. HTML produces the table headings, then a for-each loop goes through the array and prints each entry wrapped in table tags. To facilitate the real-time data streaming, the *changes* function is applied to the database. This function sets up a listener for any changes in the database state. A JavaScript library called *Socket.IO* is used to create an event handler that goes off whenever a change is received [25]. Within the event handler, new data is formatted in table tags and appended to the array, causing it to display at the end of the panel in real-time. Figure 3.4 shows the original design of the data panel.

Time	Water Temperature	pH	Air Temperature	Air Humidity
2017-05-04 15:00:01	78.0116	0	65.84	39.5
2017-05-04 14:59:01	78.0116	0	65.84	39.3
2017-05-04 14:58:01	78.0116	0	65.84	39.4

Figure 3.4: View of the Data

3.2.2 Notification Panel

The second web component is the notification panel. This collects any information a user would want to know immediately. There are two types of notifications that appear here. The first are parameter out of bounds notifications; for example, when the temperature goes above the maximum the user has set. The second type of notification is when the state of a device attached to the aquarium changes state. For example, when lights are turned on or off. Like the data panel, this makes use of the *changes* function on the notification table, and automatically updates when a new notification enters that table.

Construction of both the PHP and JavaScript versions of the notification panel are functionally similar to the data panel. The PHP version uses the *mysql_ query* call to grab the most recent rows from the notification table in the database. Then PHP prints them out onto the page in a table using the *echo* function. The JavaScript version pulls the most recent notifications as well, but also employs the *changes* function and the *Socket.IO* library to append new notifications as they are added to the database. Figure 3.5 shows an example of an entry in the original notification panel.

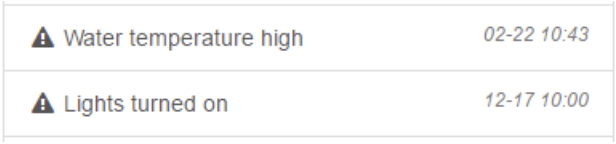


Figure 3.5: The Notification Panel

3.2.3 Visualization of the Data

The third web component is a graph that plots the data represented in the data panel. The graph presents the data in a way where it is easier to see changes over time. Currently the graph in the relational version is configured to display the last 24 readings, in order to give a better view for how parameters are changing.

The first version of the graphs are written using a mixture of PHP and JavaScript. First, PHP calls are made to pull the last 24 data points for each parameter from the database. Then the user minimum and maximum boundaries are taken from the database. The user boundaries are compared to the set of points queried from the database, which is done to determine the minimum and maximum y-axis value displayed on the graph. JQuery is used to do the actual plotting of points on the graph [26]. The second version is created exclusively with JavaScript. The *changes* function applied here allows real-time updating of the graph. When a new point comes in, it is plotted and connected on the line graph, and the oldest point is deleted. *Socket.IO* is applied here to allow for real-time data streaming. A JavaScript library called *Smoothie Charts* is used to create the graph itself [27]. *Smoothie Charts* handles scrolling the graph and deleting old points from the series as they would disappear from the chart. A key difference in the data represented by the two graphs are the two time frames they represent. The static graph shows data up until the page was generated, but the real-time graph starts with

no prior data, showing only data points from the time of page load onward. Figure 3.6 shows the original design of the data graph.

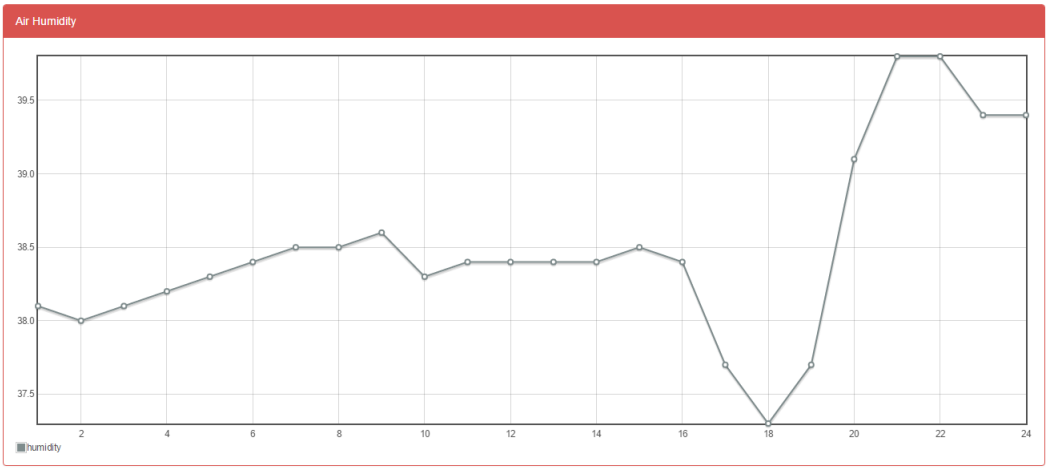


Figure 3.6: The Humidity Graph

3.3 Database Performance

3.3.1 Testing

Performance testing was done by recording the amount of time taken to complete insert, update, and read operations on various sample sizes. The sample sizes are 10,000, 100,000, 300,000, 500,000, 700,000, and 1,000,000. The first test is inserting x rows, where x is the sample size, containing a randomly generated numeric value one by one. Using this same table, two sets of x reads are made on the table. The first set of reads are sequential, reading from the first entry to the last. The second set of reads are random. To do random reads, an id value between the bounds of the first and last id listed in

the table is randomly generated, and a read is performed on that data. Then, update speeds are tested sequentially, then randomly. The test is performed similarly to the read test, with an additional step of generating new random value and overwriting the current value.

As a part of this thesis, all the code for the MySQL and RethinkDB tests have been written in PHP. The test for Mongo is written in JavaScript and completed using the Mongo shell. Before the tests start, a timestamp is recorded. Then, a loop is entered that repeats an amount of times that matches the sample size. For the MySQL and Mongo insert tests, one iteration of the loop is the generation of a random number, followed by an insert using each platforms respective query function. An additional step is taken within each iteration of the loop for RethinkDB. The value is placed into an array referred to as a document, then the *insert* function is executed on the correct table with the entire document value. At the end of each respective loop, an ending timestamp is recorded and the time taken to complete the test is calculated by subtracting the old timestamp from the new one. This calculated value is then inserted into a results table, along with the sample size tested and a date/time of the test.

The read tests run after the inserts are completed. For MySQL, the *mysql_query* function just has to run a single select statement within the loop. The id of the row selected corresponds with the value used to increment the loop, meaning all values are selected in order. The random read test is done the same way, only with an extra step of

generating a random value between zero and the test size to act as the id. For Mongo, there is a *find* function that must be used to get all values from a table, or passed an id to grab a specific entry. On the first pass, the incrementer is used as the id, on the second one, a random number is generated like the MySQL test. RethinkDB is done the same way, only using a *filter* function on the id in place of Mongo's *find* function.

The final test is the update test. It is similar to the read test, only adding another step to update the value that has been selected with a newly generated random number. For MySQL, the *mysql_ query* function will execute an update statement on the selected id. RethinkDB has a dedicated *update* function that can be added to the end of the *filter* function. Mongo has a dedicated *update* function as well, but structured differently. This function takes two parameters; what it will be matching, what fields it will be updating. At the conclusion of this test, all of the data in each of the tables are deleted. This ensures that each iteration of tests starts the same way, with a completely blank table.

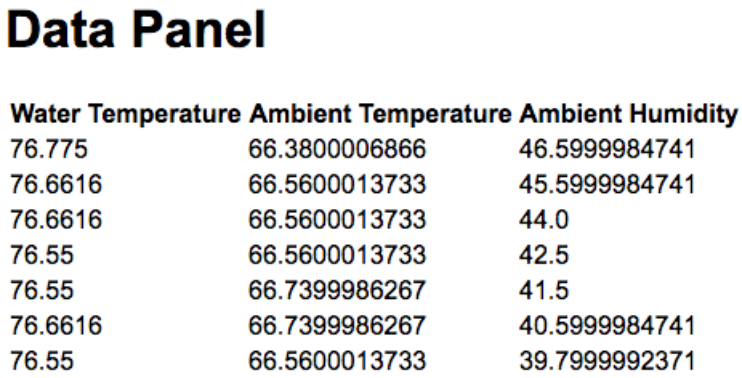
3.3.2 Monitoring

Throughout the tests, Datadog is in place to record the resource consumption of each platform. It monitors CPU usage, process memory usage, load averages, memory breakdown, available swap, disk usage, disk latency, and network traffic. This is beneficial because it is possible to see if running operations on a specific database takes more CPU or memory than another platform.

Chapter 4 - Results

4.1 Real-Time Components

These components are built on top of the RethinkDB real-time database outlined in Chapter 3. Figure 4.1 shows the new data panel. It updates in real-time, appending a new row of data to the bottom as they are entered into the database. The real-time notification panel is functionally the same, only monitoring a different table in the database.



Data Panel

Water Temperature	Ambient Temperature	Ambient Humidity
76.775	66.3800006866	46.5999984741
76.6616	66.5600013733	45.5999984741
76.6616	66.5600013733	44.0
76.55	66.5600013733	42.5
76.55	66.7399986267	41.5
76.6616	66.7399986267	40.5999984741
76.55	66.5600013733	39.7999992371

Figure 4.1: Screenshot of the Real-time view of the data

Figure 4.2 shows the real-time humidity graph. The maximum and minimum readings are displayed on the top and bottom right corners. As new rows are entered into the database, a new point representing the current humidity is plotted on the right. The graph scrolls as time passes.

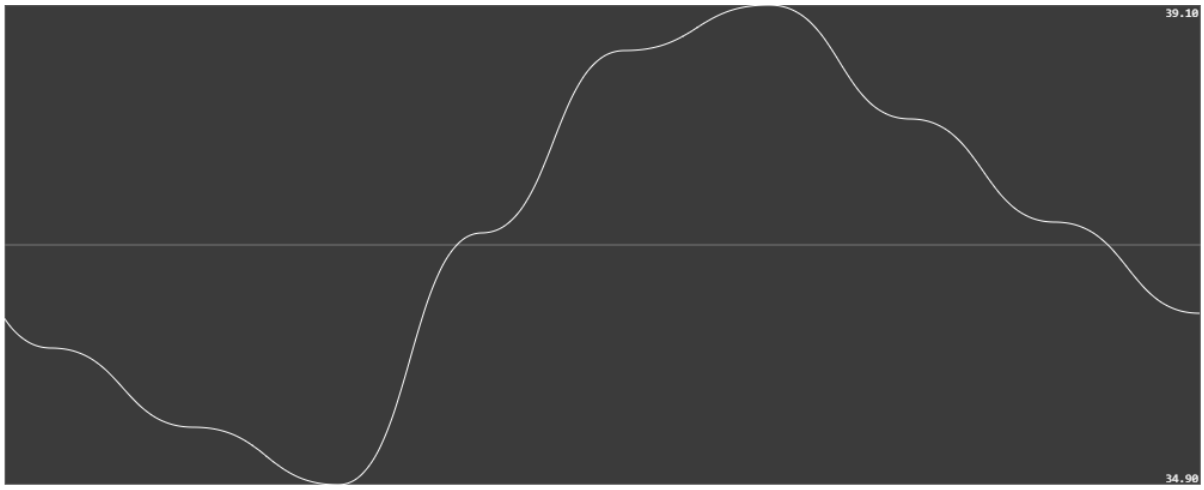


Figure 4.2: Screenshot of the Real-time graph

4.2 Query Speeds

Below are bar graphs displaying the average time taken of five runs of each test. The raw data was entered into Google Sheets and the graphs are generated using the Google Sheets chart tool. One graph shows all platforms and sample sizes used for one operation. All tests take place on the same machine. The test machine runs on the Ubuntu operating system, has a 2.4GHz dual-core processor, 4GB of RAM, and a 40GB hard drive.

4.2.1 Insert

Figure 4.3 shows that insert speeds for MySQL are comparable to MongoDB, only beating it by a few seconds for each sample size. RethinkDBs insert speed was consistently around five times slower than both MySQL and MongoDB.

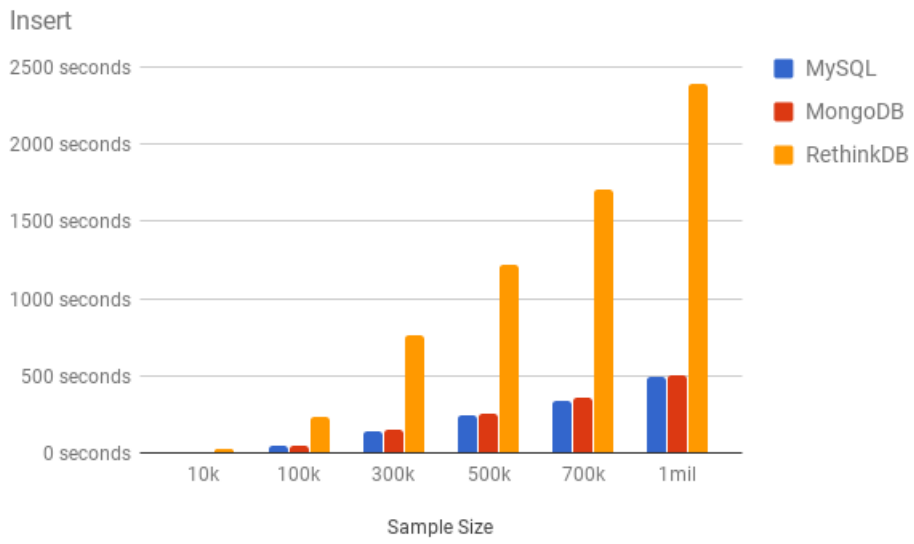


Figure 4.3: Comparison of the Insert performance

4.2.2 Sequential Read

Figure 4.4 shows that the sequential read speeds for MongoDB and MySQL are within one second of each other until the sample size reaches 500k. At sizes of 500k and above, MongoDB finished the reads a few seconds ahead of MySQL. RethinkDB is considerably slower, taking about 15 times the amount of time to complete each sample size.

4.2.3 Random Read

Figure 4.5 shows that there is no clear winner between MySQL and MongoDB for random reads. MongoDB completed the 10k, 100k, and 700k sample size tests the fastest. MySQL completed the 300k, 500k, and one million sample size the fastest. RethinkDB is the

slowest platform tested, taking on average 12-13 times longer. It should be noted that all platforms experienced a slight decrease in performance going from sequential reads to random reads, and that RethinkDB suffered the least in that regard.

4.2.4 Sequential Update

Figure 4.6 shows MySQL consistently completing all sequential update tests the fastest. MySQL finished each test roughly two times as fast as MongoDB and up to five times as fast as RethinkDB.

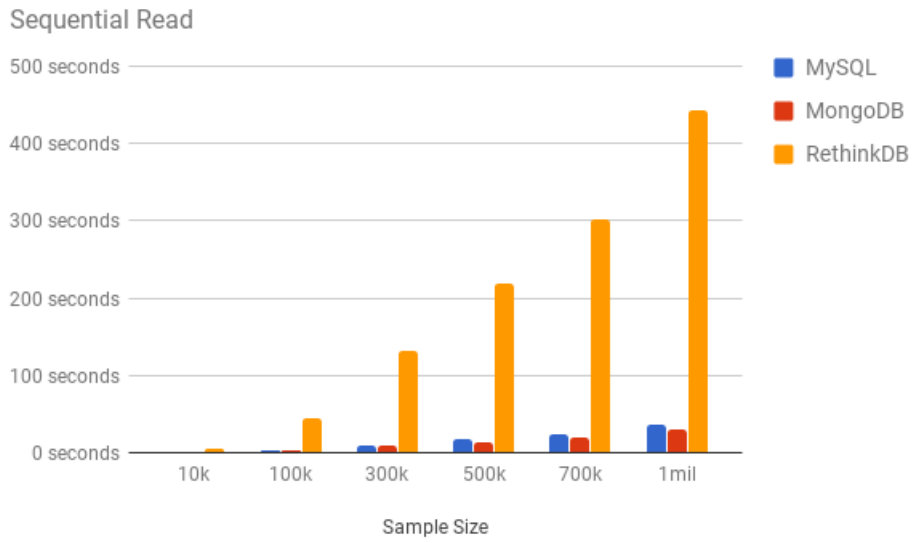


Figure 4.4: Comparison of the Sequential Read performance

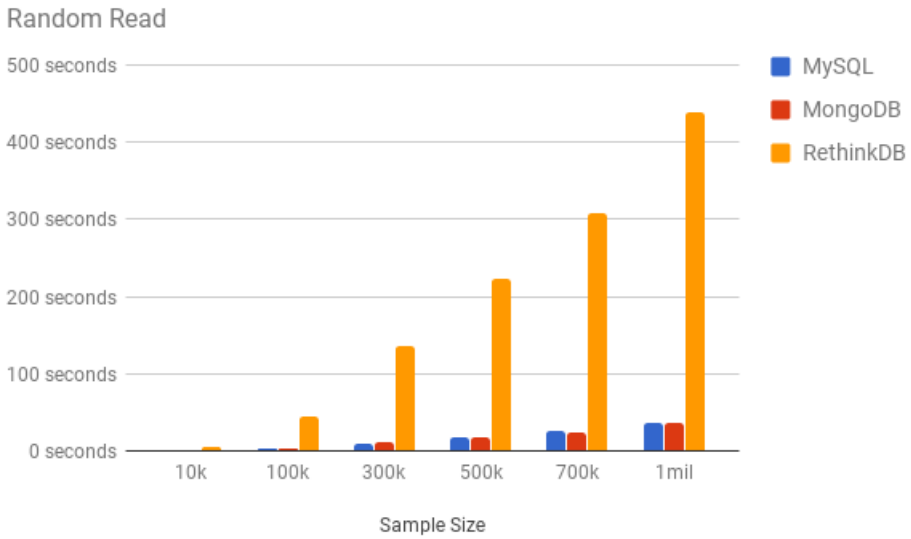


Figure 4.5: Comparison of the Random Read performance

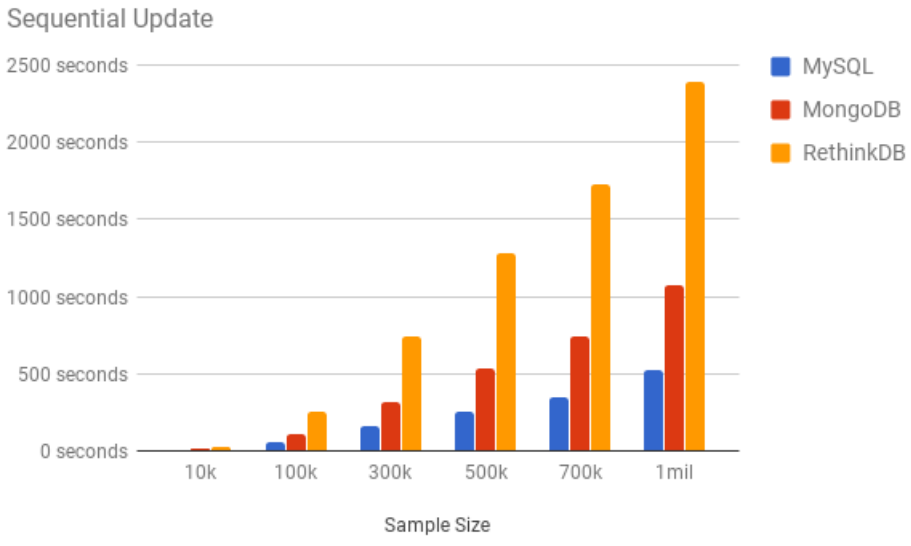


Figure 4.6: Comparison of the Sequential Update performance

4.2.5 Random Update

Moving from sequential updates to random updates caused an equivalent loss of performance across the board. So while each platform was slower at completing the random updates, the ratios of results remains the same as sequential updates. Figure 4.7 show that MySQL finished roughly two times faster than MongoDB and roughly five times faster than RethinkDB.

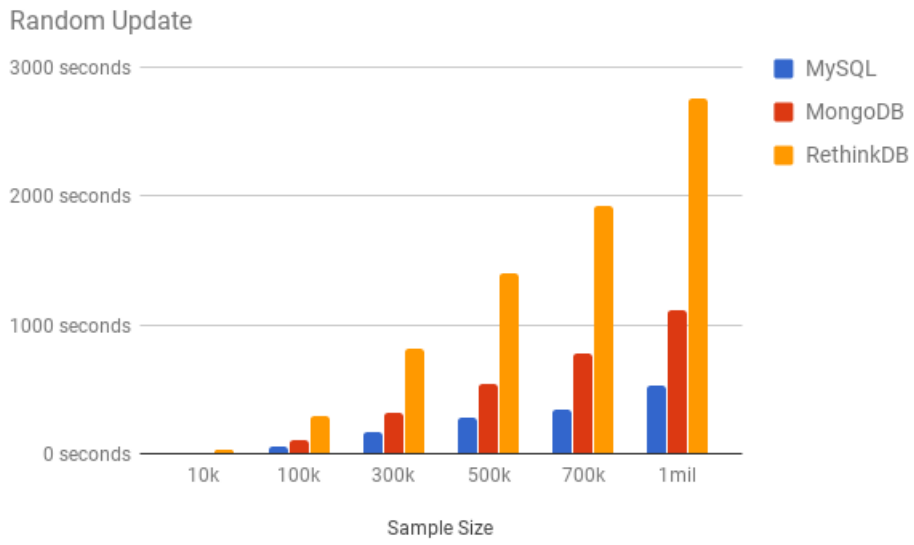


Figure 4.7: Comparison of the Random Update performance

4.3 Resource Monitoring

All of the following graphs represent each of the tests being run in order (insert, sequential read, random read, sequential update, random update) starting from the lowest sample

size and moving to the highest (10k, 100k, 300k, 500k, 700k, one million). Figure 4.8 explains what each color means in the CPU graphs.

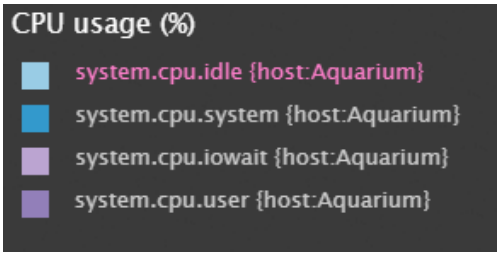


Figure 4.8: The CPU Usage Legend

4.3.1 MySQL

During the MySQL tests, shown in figure 4.9, CPU usage overall hovers around 50%. Breaking this down, the user averages 15%, the system 10%, and the remaining 25% is spent in iowait. The memory usage is a constant 1.8GB.

4.3.2 MongoDB

During the MongoDB tests, shown in figure 4.10, CPU usage overall hovers around 55%. Breaking this down, the user averages 47%, the system 7%, and iowait is less than 1%. The memory being used before and after the test is a constant 2GB. During the test the usage climbs to 2.22GB.

4.3.3 RethinkDB

During the RethinkDB tests, shown in figure 4.11, cpu usage overall hovers around 50%. Breaking this down, the user averages 25%, the system 12%, and the remaining 13% is spent in iowait. The memory usage starts at 2.07GB and steadily climbing before peaking at 2.17GB.

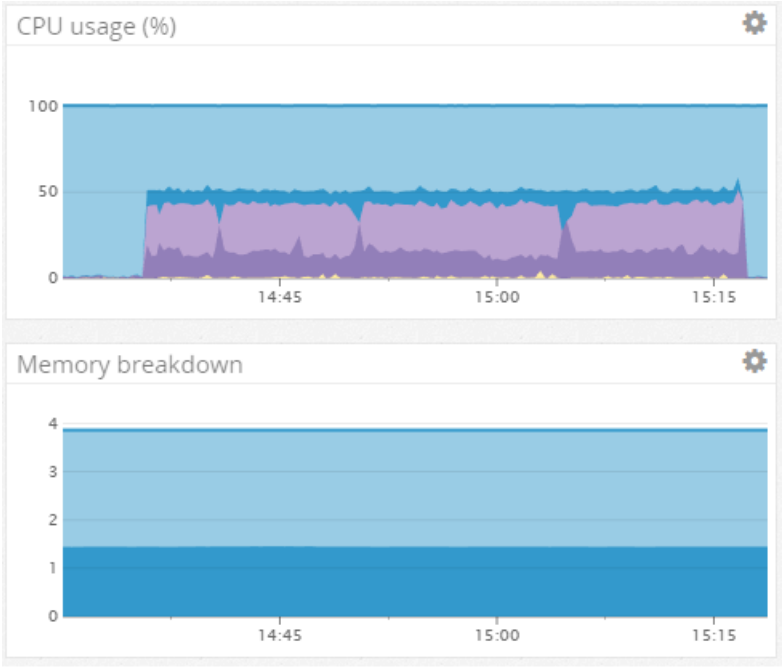


Figure 4.9: The MySQL Monitor

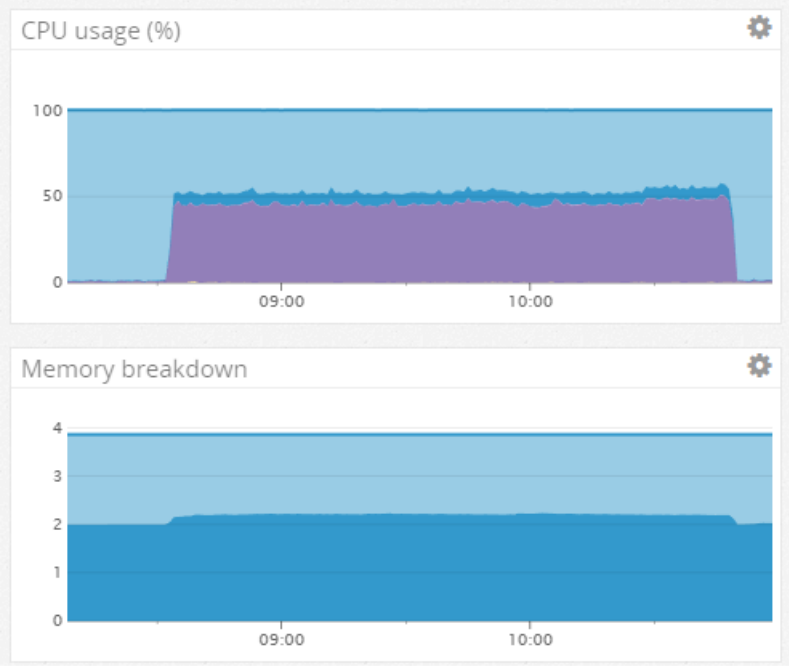


Figure 4.10: The MongoDB Monitor

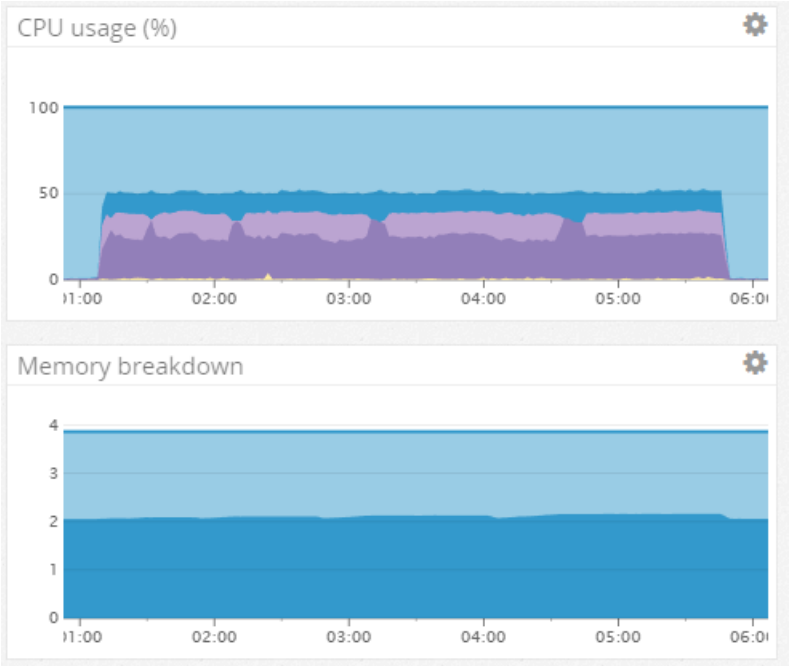


Figure 4.11: The RethinkDB Monitor

Chapter 5 - Conclusion and Future Work

When reviewing the results of the tests performed in this study, it is clear that MySQL performed the best overall in time taken to complete each test on the machine outlined in Chapter 4. Compared to MongoDB, it had comparable insert speeds, slightly worse read speeds, and almost double the update speeds. It is also very clear that RethinkDB performed the worst overall. In the best case, it took 2.5 times more time to complete a test. In the worst case, it is almost 15 times more.

This strong showing from MySQL is potentially due to the nature of the testing. In all cases, the data involved was neatly structured in a table, which is what relational database platforms excel in dealing with. In order to uncover the advantages that document stores offer, operations would have to be performed on a significantly different set of data. For example, consider a complex entity like a student being represented in a relational database. Normalizing all of the data associated with a student would yield multiple tables (contact information, classes, etc.), each with their own keys to ensure referential integrity. This data represented in MongoDB or RethinkDB can be in a single collection. To add a new student in the relational example, multiple tables have to be accessed and written to in the correct order to maintain a valid database state. Adding a new student in the document store is as simple as appending an additional document to the end of the collection. In order to fetch everything associated with a student in a

relational database, a single index lookup by id, followed by multiple range lookups of that id in other tables, would have to be made. The document store would just return the entire document associated with that id in the collection.

Similarly, a potential explanation for the overall poor comparative results of RethinkDB is the implementation used to develop the tests. The final RethinkDB tests are written in PHP. Initially, an attempt was made to implement these tests with JavaScript, however, memory issues were encountered and the tests would not complete on the second sample size of 100k records. These issues were encountered in the insert stage of implementation before any read or update tests had been written. Therefore JavaScript was abandoned and development continued with PHP. Preliminary results on insert sizes of 50k or less records in JavaScript showed significant improvement over what were the final results of the PHP testing. During the insert portion of the PHP testing, an average of approximately 300 inserts per second is shown in the RethinkDB Administration Console. During the preliminary testing for the JavaScript insert portion, an average of approximately 1000 inserts per second was observed.

Preliminary results also indicated slow update times for both MySQL and MongoDB, taking upwards of an hour or more to complete updates on 100k records (both sequentially and randomly). The initial tests failed to use an indexed field for selecting the record to update, causing a full table scan to take place for every update statement. To remedy this, update statements in MySQL used the primary key to determine what

to update, where MongoDB used the _id field. The _id field in MongoDB is the default field used to uniquely identify records.

Despite test results showing poor performance for insert, read, and update operations on different sample sizes, RethinkDB inherently excels at delivering data back to the user due to its data streaming capability. This data streaming feature is why RethinkDB is the best choice for the tables in the aquarium database that a user would want to constantly update. The naive solution for constantly receiving new aquarium data with the relational database is a periodic AJAX refresh on the page to fetch new data [28]. With the current implementation of the Arduino, new data is inserted into the database every 60 seconds. If the user sets the refresh rate to match the insertion rate, there is a chance new data is inserted only a few seconds after the AJAX refresh request is made, meaning the user has to wait almost the full period in order to receive the newest reading. In order to combat this, the refresh rate can be set to a lower value. However, doing this introduces multiple calls to the server that do not return any new data. RethinkDB coupled with *Socket.IO* uses a constantly open connection and waits to receive changes that are pushed out by the database. This ensures that each user listening receives data as soon as possible. A combination of RethinkDB and MySQL databases are the best approach for this monitoring system. RethinkDB is implemented for tables that would be constantly updated and MySQL is implemented for tables involving structured user data that may not need to be updated as often.

When analyzing the system resources consumed by each platform, the overall CPU usage did not vary much between all three. Each platform stayed steady between 50-55% throughout all operations. The RAM usage was higher than MySQL for both MongoDB and RethinkDB. This is likely explained by the caching implemented in the NoSQL platforms.

5.1 Future Work

The testing can be improved in a couple of ways. For example, all platforms can utilize batch inserts (inserting more than one entry at a time) where the current implementation inserts only one record for each iteration of the loop. Doing this should increase the overall insert speeds of each platform. Specifically, the insert speed of RethinkDB can be increased by using what they call "soft durability mode." By default, RethinkDB is in "hard durability mode" meaning that each write is committed to the disk before the client is acknowledged. Soft durability mode will acknowledge the write immediately after receiving it, before it is committed to the disk. The speed can be increased even further by entering "noreply" mode, meaning that the client will not wait for a response from the server before moving on to the next query. Horizontal scaling through sharding is natively available to both MongoDB and RethinkDB allowing for multiple machines to be involved with database operations.

Better query speeds can also be achieved by optimizing the way queries are structured. Within the tests the queries are relatively simple, but the NoSQL databases offer various ways to complete each query. For example, MonogDB offers insert, insertOne, and insertMany as different options to go about inserting data into a collection. The same can be said for the *find* function, which is MongoDB's equivalent of select in MySQL or the *get* function in RethinkDB. Multiple combinations of query types can be tried to find the most optimal way to complete the tests in this thesis.

Bibliography

[1] Philip J. Pratt. A relational approach to database design. *SIGCSE Bull. ACM SIGCSE Bulletin*, 17(1):184–201, Jan 1985.

[2] Hugh E Williams and David Lane. *Web Database Applications with PHP and MySQL: Building Effective Database-Driven Web Sites*. O'Reilly Media, Inc., 2004.

[3] Php documentation. http://php.net/docs.php/.

[4] Paolo Atzeni, Christian S. Jensen, Giorgio Orsi, Sudha Ram, Letizia Tanca, and Riccardo Torlone. The relational model is dead, SQL is dead, and I don't feel so good myself. *ACM SIGMOD Record*, 42(1):64, Jan 2013.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable. *ACM Transactions on Computer Systems*, 26(2):1–26, Jan 2008.

[6] Dorin Carstoiu, Elena Lepadatu, and Mihai Gaspar. Hbase - non sql database, performances evaluation. *International Journal of Advancements in Computing Technology*, 2(5):42–52, 2010.

[7] Abhishek1 Prasad and Bhavesh N.1 Gohil. A comparative study of nosql databases. *International Journal of Advanced Research in Computer Science*, 5(5):170–176, 2014.

[8] Pragati Prakash Srivastava, Saumya Goyal, and Anil Kumar. Analysis of various NoSql databases. In *Proceedings of 2015 International Conference on Green Computing and Internet of Things*, Delhi, India, 2015.

[9] Robin Hecht and Stefan Jablonski. Nosql evaluation: A use case oriented survey. In *Proceedings of 2012 International Conference on Cloud and Service Computing*, pages 336–341, Shanghai, China, 2011. IEEE Computer Society.

[10] Karamjit Kaur and Rinkle Rani. Modeling and querying data in NoSQL databases. In *Proceedings of 2013 IEEE International Conference on Big Data*, Santa Clara, CA, USA, 2013.

[11] Gianluca Tiepolo. *Getting started with rethinkdb.* Packt Publishing Limited, 2016.

[12] Ben Kao and Hector Garcia-Molina. An overview of real-time database systems. *Real Time Computing NATO ASI Series*, 127:261–282, 1994.

[13] Jan Lindström. Real time database systems. *Wiley Encyclopedia of Computer Science and Engineering*, 2008.

[14] Raul Barbosa. An essay on real-time databases. *Chalmers University of Technology*, 2007.

[15] Rajib Mall. *Real-Time Systems: Theory and Practice.* Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2009.

[16] Kam-Yiu Lam and Tei-Wei Kuo. *Real-time Database Systems : Architecture and Techniques.* Number SECS 593 in The Kluwer International Series in Engineering and Computer Science. Springer, 2001.

[17] Mohamed A Mohamed, Obay G Altrafi, and Mohammed O Ismail. Relational vs. nosql databases: A survey. *International Journal of Computer and Information Technology*, 3(03):598–601, 2014.

[18] AmitPal Priyanka. A review of nosql databases, types and comparison with relational database. *International Journal of Engineering Science*, 4963, 2016.

[19] Jagdev Bhogal and Imran Choksi. Handling big data using nosql. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, pages 393–398, Gwangju, Korea, 2015. IEEE.

[20] Zachary Parker, Scott Poe, and Susan V Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*. ACM, 2013.

[21] Xinkui Zhao, Jianwei Yin, Chen Zhi, Pengxiang Lin, Shichun Feng, Hao Wu, and Zuoning Chen. monbench: A database performance benchmark for cloud monitoring system. In *2015 IEEE International Conference on Cluster Computing*, pages 523–524. IEEE, 2015.

[22] Datadog docs. http://docs.datadoghq.com/.

[23] Arduino. https://www.arduino.cc/.

[24] Javascript reference. https://www.javascript.com/.

[25] Socket.io docs. https://socket.io/docs/.

[26] jquery api documentation. http://api.jquery.com/.

[27] Smoothie charts. http://smoothiecharts.org/.

[28] Ajax. https://developer.mozilla.org/en-US/docs/AJAX.

[29] Konstantina Papagiannaki, Rene Cruz, and Christophe Diot. Network performance monitoring at small time scales. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 295–300, Miami Beach, FL, USA, 2003. ACM.

Vita

Devin Sink was born in Lexington, North Carolina in September 1990. He graduated from West Davidson High School in 2008, then entered Davidson County Community College before transferring to Appalachian State University in 2013. He graduated in December of 2015 with a Bachelor of Science in Computer Science. He decided to immediately continue his studies at Appalachian by pursuing a Master of Science in Computer Science the following semester. He then graduated with the Master of Science in Computer Science in August of 2017.